# METHOD AND SYSTEM FOR PROVIDING AN AUTHORIZATION FRAMEWORK FOR APPLICATIONS

## COPYRIGHT NOTICE



## BACKGROUND OF THE INVENTION

The invention disclosed herein relates generally to systems and methods for providing an authorization framework for applications. More particularly, the present invention provides for declaratively specifying authorization enforcement points and associating them with permission classes and subclasses by using declarations that map constants, local variables, or instance variables to permission classes and subclasses such as Java Permission, PermissionFactory, PrivilegedAction and PrivilegedActionFactory classes.

Providing security measures that allow granularity involves establishing a complex set of relationships between principals and permissions for authorizations to run or use certain processes in applications. A "principal" is an entity in the computer system to which permissions are granted, and is also called a "target application" herein, which contains classes of objects, or "target classes." Examples of principals include processes, objects and threads. A "permission" is an authorization by the computer system that allows a principal to perform a particular action or function. In many cases, the principals also involve user-oriented processes,

such as a user interface providing authorized access to a database, wherein the permission to be granted is to allow the user interface access to the database so the user may have access.

There are several systems implemented for use with Java "permission" classes, such as the system described in U.S. Patent No. 6,047,377 to Gong ("Gong"). Gong describes a method and apparatus for establishing and maintaining complex security rules. The security rules are established through the use of permission classes that have the ability to inherit attributes and methods. For example, a permission super class is established that defines an interface to a validation method. A permission subclass may then be created which provides an implementation of the validation method. When invoked, the validation method indicates whether a given permission represented by one object belonging to a permission class encompasses the permission represented by another object belonging to a permission class. Classes are also provided for grouping permissions into sets, and for establishing protection domains for classes of objects. However, using the system described in Gong, in order to introduce new permission subclasses, or to change permission subclasses, developers are still required to perform considerable programming, or re-programming, to change the references to those subclasses for each authorization enforcement point in a target application containing those references.

## BRIEF SUMMARY OF THE INVENTION

The present invention addresses, among other things, the problems discussed above with respect to applications' granting of permissions to target applications, such as target classes, processes, objects and threads.

The present invention provides a system and method, or framework, for declaratively specifying authorization enforcement points and associating them with permission

subclasses. Authorization enforcement code is inserted into one or more of the target classes of the target application according to the authorization enforcement points

For example, some embodiments are for use in the Java Authentication and Authorization API (JAAS). In those embodiments, the invention provides for declarative specification of authorization enforcement points, which are associated with the java.security.Permission subclass (or a PermissionFactory), and/or a java.security.PrivilegedAction implementation (or a PrivilegedActionFactory). The system fully exposes the flexibility and extensibility of the JAAS framework by providing application deployers or developers the ability to introduce their own permission subclasses and/or PrivilegedAction implementations for evaluation at the authorization enforcement points in a non-programmatic manner.

In the present invention, for the Java/JAAS embodiment using a SubjectFactory, PermissionFactory, and a PriviliegedAction factory, a declaration specifies the PermissionFactory, and/or the PrivilegedActionFactory, capable of producing the appropriate permission and/or PrivilegedAction implementation, which may also be based on run-time conditions. The declaration also specifies a SubjectFactory capable of determining the javax.security.auth.Subject instance which will be associated with the access control context. Declarations may map constants, local variables, or instance variables to Java Permission, PermissionFactory, PrivilegedAction and PrivilegedActionFactory properties.

Authorization enforcement points are not limited to a "container" scope -- any class may be instrumented with authorization enforcement code. It is not necessary to have source code for the target class.

## BRIEF DESCRIPTION OF THE DRAWINGS

The invention is illustrated in the figures of the accompanying drawings which are meant to be exemplary and not limiting, in which like references are intended to refer to like or corresponding parts, and in which:

Fig. 1 is a block diagram illustrating the overall structure of some embodiments of the present invention implemented using the Java Authentication and Authorization Service API (JAAS) is shown;

Fig. 2 illustrates a method for an example target class of Fig. 1;

Fig. 3 illustrates an example deployment descriptor for the target application of Fig. 1 is shown;

Fig. 4 is a flow diagram illustrating a method performed by the system represented in Fig. 1;

Fig. 5 is an example of the instrumented target class according to the example implementation of the invention of Figs. 2 and 3; and

Fig. 6 is the byte code representation of the instrumented target class code of Fig. 5, with exemplary byte code insertion implementation code shown at the enforcement points.

## DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Preferred embodiments of the invention are now described with reference to the drawings. In accordance with the invention, and with reference to Fig. 1, a block diagram illustrates the overall structure of some embodiments implemented using the Java Authentication and Authorization Service API (JAAS) The JAAS provides flexible and scalable mechanisms for securing client-side and server-side Java applications. The JAAS is pluggable and stackable, and lets developers incorporate standard security mechanisms like Solaris NIS, Windows NT,

lightweight access directory protocol (LDAP), Kerberos, and others, into applications in a consistent, configurable way.

The system of the present invention directly instruments target class binaries at deployment time or at run time through a class loader based on deployment descriptor settings. Instrumentation of Java bytecode comprises inserting a short sequence of bytecode at designated points in the Java classes of an application. Such a technique is commonly used to facilitate runtime analysis of those classes, which involves mostly profiling, monitoring, or other introspection tasks. However, the present invention uses the technique to insert its instrumentation code (100 in Fig. 1).

There are two types of instrumentation, static and dynamic. Static instrumentation of code can occur during or after compilation, but dynamic instrumentation can take place at runtime, which includes after the target application is loaded in a memory for execution. Runtime class instrumentation is performed through a class preprocessor 50, whereby a class preprocessor 50 inserts instrumentation code 100 at the required places in the target Java classes 102 just before it is loaded by a Java virtual machine. The class preprocessor 50 works in conjunction with an instrumentation class loader 106.

The instrumentation code 100, which may be derived or compiled from an instrumentation application 108, contains references to a custom authorization deployment descriptor 104. The deployment descriptor 104 is a file that contains initialization parameters for the instrumentation code 100. Declarative specifications that map constants, local variables, or instance variables to Permission, PermissionFactory, PrivilegedAction, and PrivilegedActionFactory properties are stored in the deployment descriptor 104. The instrumentation code 100 initializes the variables, and calls the privilege action procedures that

perform the authorizations. The constants in the deployment descriptor 102 may be easily defined or changed, even at runtime, after the instrumentation code 100 has been defined or compiled. Insertion of the instrumentation code 100 into the target class code 102 may occur as late as during deployment or at runtime of the target application 102, which may included after the application is loaded into a memory for execution, at which time the constants are read to determine the correct authorization point values to insert into the instrumentation code 100.

With reference to Fig. 2, the code for an example method 102a of target class 102 is shown. The class is named "ProfileService." In this simple illustration, the method 102a only has one line of code, which is to print the line "*** ProfileService.getProfile entry ***" on the screen while it is executing.

With reference to Fig. 3, example code 104a for a deployment descriptor 104 is shown. The target is defined in the first line as the "ProfileService" class of Fig. 2. In the second line of the descriptor, the subject factory class is called "ThreadSubjectFactory." On line 3, the descriptor starts for the method "getProfile" (which is the method shown in 102a of Fig. 2). The required permission class is named on line 4 as "com.ibm.resource.security.auth.ProfilePermission", and the various properties of the com.ibm.resource.security.auth.ProfilePermission permission class are defined in lines 5-7. A permission class is a class for representing access to a system resource, which is, in this case, the getProfile method of the ProifleService target class. The name of the class is typically a pathname of a file (or directory). Permission class objects typically include an "actions" list that lists the actions that are permitted for the object. The actions list (such as "read, write") specifies which actions are granted for the specified file (or for files in the specified directory). For example, one of the listed actions that may be authorized for the requiredPermission class object

com.ibm.resource.security.auth.ProfilePermission is an action called "ReadAction," which is used to read the properties defined in lines 5-7 from a profile database indicated by the com.ibm.resource.security.auth.ProfilePermission class object. Lines 9-11 shows an example descriptor for the privileged action ReadAction along with the attributes for that action.

One method that is typically implemented in a permission class is an "implies" method to compare permissions. For example, "permission p1 implies permission p2" would mean that if a user is granted permission p1, the user is naturally granted permission p2. In the case of the class object com.ibm.resource.security.auth.ProfilePermission, if that object has a WriteAction permission, and permission is granted to the WriteAction action, the implies method may imply permission to the ReadAction permission.

With reference to Fig. 4, a flow diagram is shown illustrating the steps performed by the system and method represented in Fig. 1. After a target class 102 is coded or identified, step 350, the instrumentation application is coded or changed for the target class 102, step 352. The deployment descriptor is coded, step 354. If necessary or preferred by the developer, the instrumentation code may then be compiled into the instrumentation code 100, step 356. Otherwise, the instrumentation code 100 may remain in scripted form depending on the type of system in which the invention is being implemented. At either runtime, or sometime before runtime, of the target application containing the target class 102, class instrumentation is performed using the class preprocessor 50, whereby a class preprocessor 50, in conjunction with the instrumentation class loader 106, inserts instrumentation code 100 at the required places in the target Java class 102 just before it is loaded by a Java virtual machine, step 358. The target application may then be executed, performing the instrumentation code 100 at the appropriate times during execution, step 360.

With reference to Fig. 5, an example of the instrumented code 102b class is shown according to the example implementation of the invention of Figs. 2 and 3, which is the state of the method code 102a from Fig. 2 after instrumentation of instrumentation code 100 by the class preprocessor 50. As mentioned in the comment section on lines 4-5 of the code, although the authorization enforcement code in the example is actually inserted as byte code, for the purposes of illustration, the Java code version of the instrumented code 102b is shown in Fig. 5. As can be seen, in the instrumented part of code 102b, the preprocessor 50 has read the deployment descriptor code 104a, and added the appropriate instrumentation code 100 according to the parameters in the deployment descriptor code 104a. The instrumentation code added to the target class code 102a in this example 102b creates permissionFactory and privilgedActionFactory objects, sets the properties needed to read the profile database by running the a setPropery method of those objects, and executes a getPrivilegeAction method which executes the ReadAction privilege action of the instrumentation code 100.

With reference to Fig. 6, the byte code representation of the instrumented target class code 102b of Fig. 5 is shown. In addition to the instrumented byte code, the byte code insertion implementation code 600 is shown. In this embodiment, the code may be created by the system according to the code 104a of the deployment descriptor 104. The example code shown is compliant with the publicly available byte code engineering laboratory format. However, those skilled in the art would recognize that the byte insertion code may be implemented using a variety languages and formats, including C, C++, or assembly languages.

While the invention has been described and illustrated in connection with preferred embodiments, many variations and modifications as will be evident to those skilled in this art may be made without departing from the spirit and scope of the invention, and the

invention is thus not to be limited to the precise details of methodology or construction set forth above as such variations and modification are intended to be included within the scope of the invention.

5